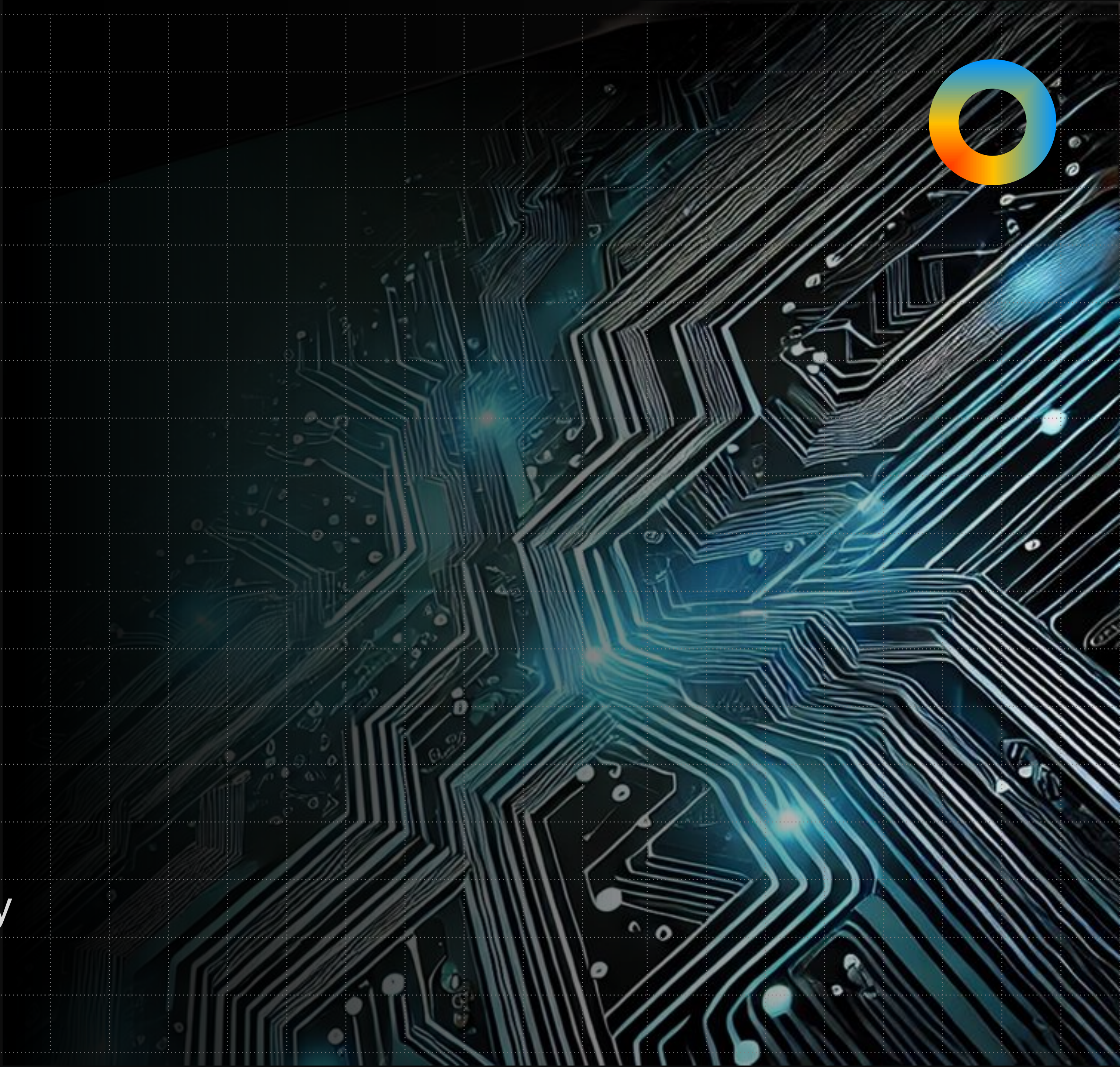
Python Programming

Flow Control

Dr. Chun-Hsiang Chan
Department of Geography
National Taiwan Normal University



Outlines

- Conditions
- If… Else…
- For Loop
- While Loop
- Pass, Continue, Break
- Try… Except…

Conditions

- Usually, we need to use ”**conditions**” to avoid occurring some cases or situations. Sometimes, we just want to classify all items into several categories by following some rules.

```
# we can simply use logical conditions to solve this
a, b = [3, 5]
a == b # equal
a != b # not equal
a < b # less than
a <= b # less than or equal to
a > b # greater than
a >= b # greater than or equal to
```

Conditions

- Python relies on indentation (whitespace/ tab at the beginning of a line) to define scope in the code.
- Other programming languages often use curly-brackets for this purpose.

```
# simple condition with if
a, b = [3, 5]
if a == b:
    print("a is equal to b")
```

The whitespace/ tab here stands for indentation.
I usually use “tab” button for indentation because it is much simpler and makes consistent to other indentations.

https://www.w3schools.com/python/python_conditions.asp

If… Else…

- In most cases, only one “**if**” cannot satisfy our real-world problems; therefore, here, I introduce other items – “**elif** and **else**”.

```
# simple condition with if, elif, else
a, b = [3, 5]
if a == b:
    print("a is equal to b")
elif a > b:
    print("a is larger than b")
else:
    print("a is smaller than b")
```

If… Else…

- Are you satisfied with the functionality of “**if…else**”? I do not think so because some cases require two or more conditions in a single procedure. For example, how to extract all postmenopausal women with single-line code?

```
# two or more conditions
a, b, c, d = [3, 5, 51, 500]
if a == b and a < d:
    print("situ 1")
elif c < b or c > d:
    print("situ 2")
elif not c < b:
    print("situ 3")
```

If… Else…

- In addition to multiple condition, we can leverage nested conditions for complicated problems or situations.

```
# nested conditions
a, b, c, d = [3, 5, 51, 500]
if a == b:
    if a < d:
        print("situ 1")
elif c < b or c > d:
    print("situ 2")
elif c > b:
    pass # do nothing
```

Lab Practice 1 (conditions)

- Design a function for determining your GPA of each subject.
- Please try these cases:
  1) 92
  2) 60
  3) 2
  4) 0
  5) 102
  6) -5

  # notice: your code needs to avoid incorrect inputs

Letter	Range	Grade Point
A+	90-100	4.3
A	85-89	4
A-	80-84	3.7
B+	77-79	3.3
B	73-76	3
B-	70-72	2.7
C+	67-69	2.3
C	63-66	2
C-	60-62	1.7
D	50-59	1
E	1-49	0
X	0	0

Lab Practice 1 (conditions)

- The input and output of the function is as follows:

```
# Format
def GPA(score):
    # annotation
    …
    …
    …
    return gpa
```

- The return value of the function is **GPA**.
- The data type of function output is **string**.

For Loops

```
for i in range(end): # start from 0
```

- In Python, we have two loop functions for item-wise iteration.
- For example, we want to print all numbers ranging from 0 to 100, individually.

```
# for loop
for i in range(100):
    print(i) # does it iterate to 100? If not, how can you fix it

# for loop with a condition
for i in range(100):
    if i/10==0:
        print(i)
```

For Loops

- In some cases, we can directly iterate with other approaches.

```
# for loop with a list
scorels = [78, 80, 100, 89, 50, 65, 70]
for i in scorels:
    print(i) # what does it iterate and output?

# for loop with a string
for i in "taiwan":
    print(i) # what does it iterate and output?
```

For Loops

- If you have some conditions, and then you need other tools.

```
# for loop with conditions and rules
scorels = [78, 80, 100, 89, 50, 65, 70]
for i in scorels:
    if i < 60:
        print(i, "you are failed in this subject!")
    elif i > 100 or i < 0:
        break
    else:
        pass
# change the scores and observe the functionality of break and pass
```

For Loops

- One loop cannot satisfy our requests, ….
- Therefore, we introduce another approach – **nested loop**.

```
# nested for loop
for i in range(10):
    # print(i)
    for j in range(10):
        print(i, j)
```

```
# nested for loop
(1) i = 0 then j = 0 to 9, respectively
(2) i = 1 then j = 0 to 9, respectively
(3) i = 2 then j = 0 to 9, respectively
(4) i = 3 then j = 0 to 9, respectively
(5) …
(6) …
(7) …
(8) …
```

For Loops

- How can we change the iteration way?

```
for i in range(start, end, hopping_step):
```

```
# hopping with 5 step
for i in range(0, 100, 5):
    print(i)

# reverse hopping
for i in range(100, 0, -10):
    print(i)

# observe the regularity
```

While Loops

- While loops are very different from for loop because of their nature. For example, for loop has a variable that could change in each iteration; however, while loop does not require to do so.
- Without using a changeable variable, how does while loop work?
- And what is the benefit of while loop compared to for loop?
- **Think about this**.

While Loops

while ending_condition:

- At the beginning, we demonstrate a simple example…

```
# while loop
i = 0
while i<10:
    print(i)
    i += 1 # equals to i = i + 1
```

```
# infinite while loop
i = 0
while 1:
    if i > 10:
        break # stop iteration
    else:
        print(i)
        i += 1
        continue # keep iteration
```

While Loops

- How about nested while loop?

```
# nested while loop
i = 0
while i < 5:
    j = 0
    while j < 5:
        print(i, j)
        j += 1
    i += 1 # equals to i = i + 1
```

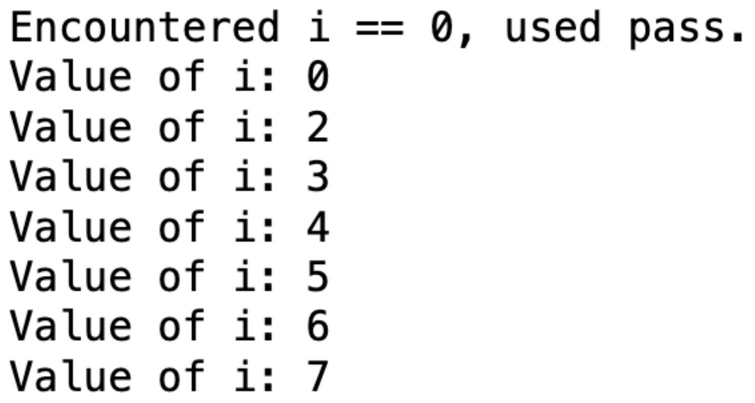
Pass, Continue, Break

- In Python, pass, continue, and break are control statements used to manage the flow of your code—particularly within loops.
- **pass**: A placeholder statement that does nothing and is often used to satisfy syntactical requirements when no code needs to be executed yet.
- **continue**: Skips the rest of the current iteration in a loop and proceeds directly to the next iteration.
- **break**: Immediately terminates the enclosing loop and proceeds to execute the next statement after the loop.

Pass, Continue, Break

```python
for i in range(10):
    # 'pass' does nothing but is syntactically required here
    if i == 0:
        pass
        print("Encountered i == 0, used pass.")
    # skip further processing in this iteration
    elif i == 1:
        continue
        print("Encountered i == 0, used continued.")
    # exit the loop entirely
    elif i == 8:
        break
    print(f"Value of i: {i}")
```

Results:



```
Encountered i == 0, used pass.
Value of i: 0
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
Value of i: 6
Value of i: 7
```

Lab Practice #2 (for and while)

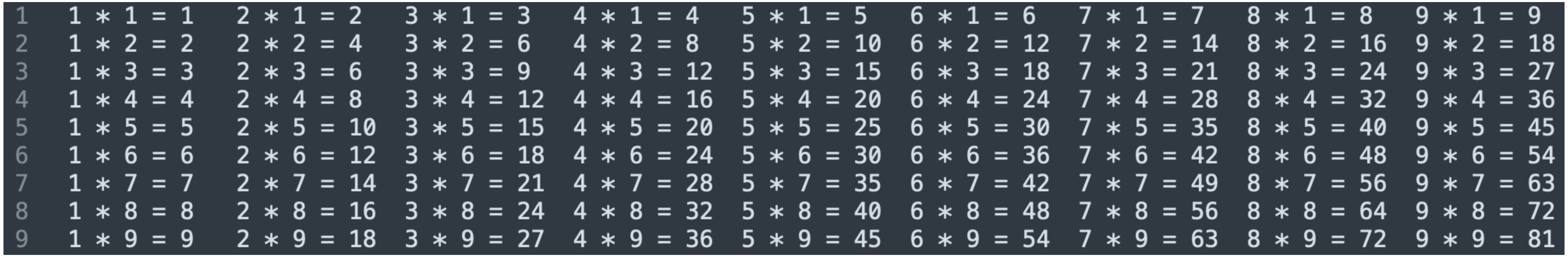
- Design a function that can produce the following results.

```
# Format
def crosstable(number):
    # annotation
    …
    …
    …
    return None
```

- There is no return in this function.
- The cross table should be directly printed.

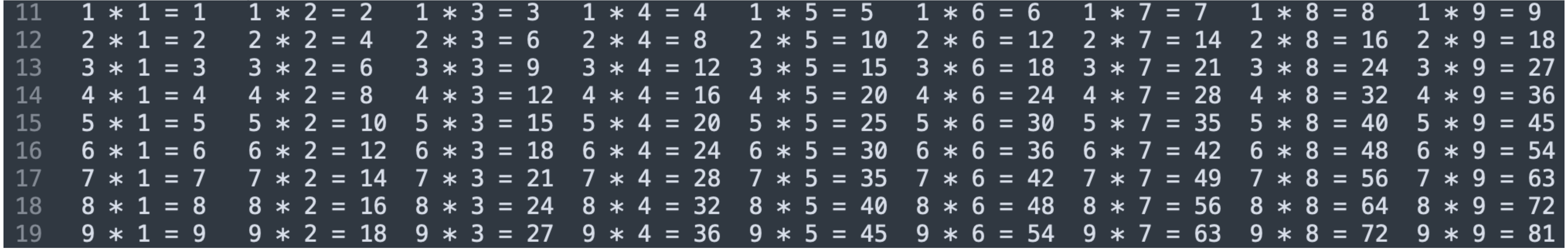
Lab Practice #2 (for and while)

- Make a 9 x 9 multiplication table.



```
1 * 1 = 1   2 * 1 = 2   3 * 1 = 3   4 * 1 = 4   5 * 1 = 5   6 * 1 = 6   7 * 1 = 7   8 * 1 = 8   9 * 1 = 9
1 * 2 = 2   2 * 2 = 4   3 * 2 = 6   4 * 2 = 8   5 * 2 = 10  6 * 2 = 12  7 * 2 = 14  8 * 2 = 16  9 * 2 = 18
1 * 3 = 3   2 * 3 = 6   3 * 3 = 9   4 * 3 = 12  5 * 3 = 15  6 * 3 = 18  7 * 3 = 21  8 * 3 = 24  9 * 3 = 27
1 * 4 = 4   2 * 4 = 8   3 * 4 = 12  4 * 4 = 16  5 * 4 = 20  6 * 4 = 24  7 * 4 = 28  8 * 4 = 32  9 * 4 = 36
1 * 5 = 5   2 * 5 = 10  3 * 5 = 15  4 * 5 = 20  5 * 5 = 25  6 * 5 = 30  7 * 5 = 35  8 * 5 = 40  9 * 5 = 45
1 * 6 = 6   2 * 6 = 12  3 * 6 = 18  4 * 6 = 24  5 * 6 = 30  6 * 6 = 36  7 * 6 = 42  8 * 6 = 48  9 * 6 = 54
1 * 7 = 7   2 * 7 = 14  3 * 7 = 21  4 * 7 = 28  5 * 7 = 35  6 * 7 = 42  7 * 7 = 49  8 * 7 = 56  9 * 7 = 63
1 * 8 = 8   2 * 8 = 16  3 * 8 = 24  4 * 8 = 32  5 * 8 = 40  6 * 8 = 48  7 * 8 = 56  8 * 8 = 64  9 * 8 = 72
1 * 9 = 9   2 * 9 = 18  3 * 9 = 27  4 * 9 = 36  5 * 9 = 45  6 * 9 = 54  7 * 9 = 63  8 * 9 = 72  9 * 9 = 81
```

- Another style.



```
1 * 1 = 1   1 * 2 = 2   1 * 3 = 3   1 * 4 = 4   1 * 5 = 5   1 * 6 = 6   1 * 7 = 7   1 * 8 = 8   1 * 9 = 9
2 * 1 = 2   2 * 2 = 4   2 * 3 = 6   2 * 4 = 8   2 * 5 = 10  2 * 6 = 12  2 * 7 = 14  2 * 8 = 16  2 * 9 = 18
3 * 1 = 3   3 * 2 = 6   3 * 3 = 9   3 * 4 = 12  3 * 5 = 15  3 * 6 = 18  3 * 7 = 21  3 * 8 = 24  3 * 9 = 27
4 * 1 = 4   4 * 2 = 8   4 * 3 = 12  4 * 4 = 16  4 * 5 = 20  4 * 6 = 24  4 * 7 = 28  4 * 8 = 32  4 * 9 = 36
5 * 1 = 5   5 * 2 = 10  5 * 3 = 15  5 * 4 = 20  5 * 5 = 25  5 * 6 = 30  5 * 7 = 35  5 * 8 = 40  5 * 9 = 45
6 * 1 = 6   6 * 2 = 12  6 * 3 = 18  6 * 4 = 24  6 * 5 = 30  6 * 6 = 36  6 * 7 = 42  6 * 8 = 48  6 * 9 = 54
7 * 1 = 7   7 * 2 = 14  7 * 3 = 21  7 * 4 = 28  7 * 5 = 35  7 * 6 = 42  7 * 7 = 49  7 * 8 = 56  7 * 9 = 63
8 * 1 = 8   8 * 2 = 16  8 * 3 = 24  8 * 4 = 32  8 * 5 = 40  8 * 6 = 48  8 * 7 = 56  8 * 8 = 64  8 * 9 = 72
9 * 1 = 9   9 * 2 = 18  9 * 3 = 27  9 * 4 = 36  9 * 5 = 45  9 * 6 = 54  9 * 7 = 63  9 * 8 = 72  9 * 9 = 81
```

Try… Except…

- Sometimes, our code may encounter unexpected or unknown errors.
- For example, if we need to read large amounts of data from multiple files and then preprocess it, it’s important to use a "try... except..." block to handle potential issues and maintain reliable execution.

Try… Except…

- **try**: Encloses a block of code that may raise exceptions during execution.
- **except**: Catches and handles specific exceptions that occur within the corresponding try block.
- **else**: Executes only if no exceptions were raised in the try block.
- **finally**: Runs unconditionally after the try (and any except blocks), whether or not an exception was raised.

Try… Except…

- Here is an example for {**try**, **except**, **finally**}…

```
try:
    # some code that might raise an exception
    print(x)
    pass
# caught an exception
except Exception as e:
    print("Caught an exception:", e)
# print the following text anyway
finally:
    print("This always executes, regardless of whether an exception occurred or not.")
```

Try… Except…

- Here is an example for {**try**, **except**, **else**}…

```
try:
    # some code that might raise an exception
    print(x)
    pass
# caught an exception
except Exception as e:
    print("Caught an exception:", e)
# print the following text anyway
else:
    print("This always executes, regardless of whether an exception occurred or not.")
```

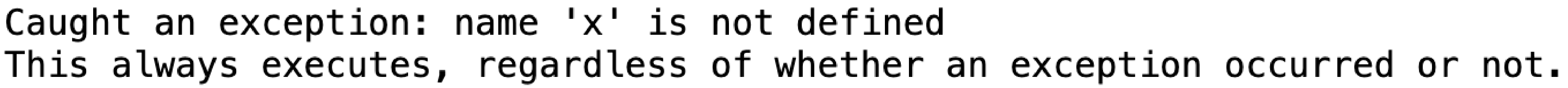
Try… Except…

- Here is an example for {**try**, **except**, **else**}…

```
try:
    # some code that might raise an exception
    print(x)
    pass
# caught an exception
except:
    print("Caught an exception")
# print the following text anyway
else:
    print("This always executes, regardless of whether an exception occurred or not.")
```

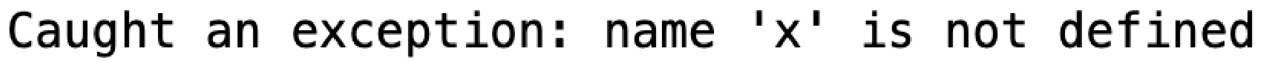
Try… Except…

- Here is an example for {**try**, **except**, **finally**}…



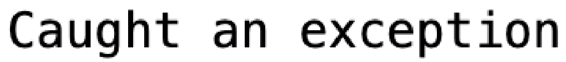
```
Caught an exception: name 'x' is not defined
This always executes, regardless of whether an exception occurred or not.
```

- Here is an example for {**try**, **except**, **else**}…



```
Caught an exception: name 'x' is not defined
```

- Here is an example for {**try**, **except**, **else**}…



```
Caught an exception
```

Lab Practice #3 Check Reciprocal

- Design a function that can produce the following results.

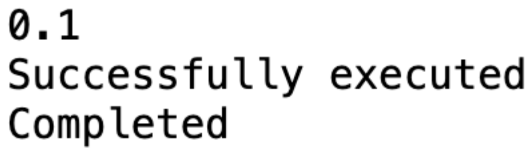
```
# Format
def checkReciprocal(number):
    # annotation
    ...
    ...
    ...
    return reciprocal
```

- Input should be a real number.
- If there is an exception, then print the exception information.
- If there is no exception, then print "Successfully executed."
- When the code is completed, it has to print "Completed."

Lab Practice #3 Check Reciprocal

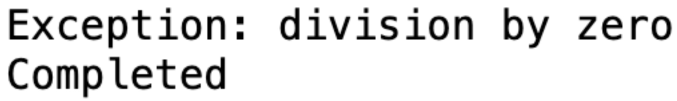
- Here are some examples.

checkReciprocal(10)



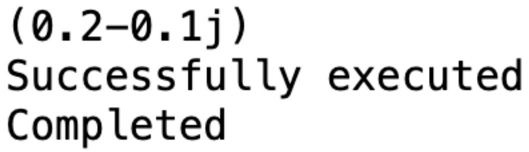
```
0.1
Successfully executed
Completed
```

checkReciprocal(0)



```
Exception: division by zero
Completed
```

checkReciprocal(2j+4)



```
(0.2-0.1j)
Successfully executed
Completed
```

The End

Thank you for your attention!

Email: chchan@ntnu.edu.tw

Website: https://toodou.github.io/

STDS Lab
Est. 2022